(12) UK Patent (19) GB (11) 2 325 539 (13) B

(54) Title of Invention

Reusing code in object-oriented program development

(51) INT CL^7; G06F 9/44

(21) Application No
9806329.0

(22) Date of filing
26.03.1998

(30) Priority Data

(31) 08846869

(32) 01.05.1997

(33) US

(43) Application published
25.11.1998

(45) Patent published
23.01.2002

(52) Domestic classification
(Edition T)
G4A APL

(56) Documents cited
EP0406028 A2

(58) Field of search

As for published application
2325539 A *viz:*
UK CL(Edition P) G4A APL
INT CL^6 G06F
Online: WPI
updated as appropriate

(72) Inventor(s)
Christina Lau

(73) Proprietor(s)
International Business
Machines Corporation

(Incorporated in USA -
New York)

Armonk
New York 10504
United States of America

(74) Agent and/or
Address for Service
C Boyce
IBM United Kingdom Limited
Intellectual Property Dept
Hursley Park
Winchester
Hampshire
SO21 2JN
United Kingdom

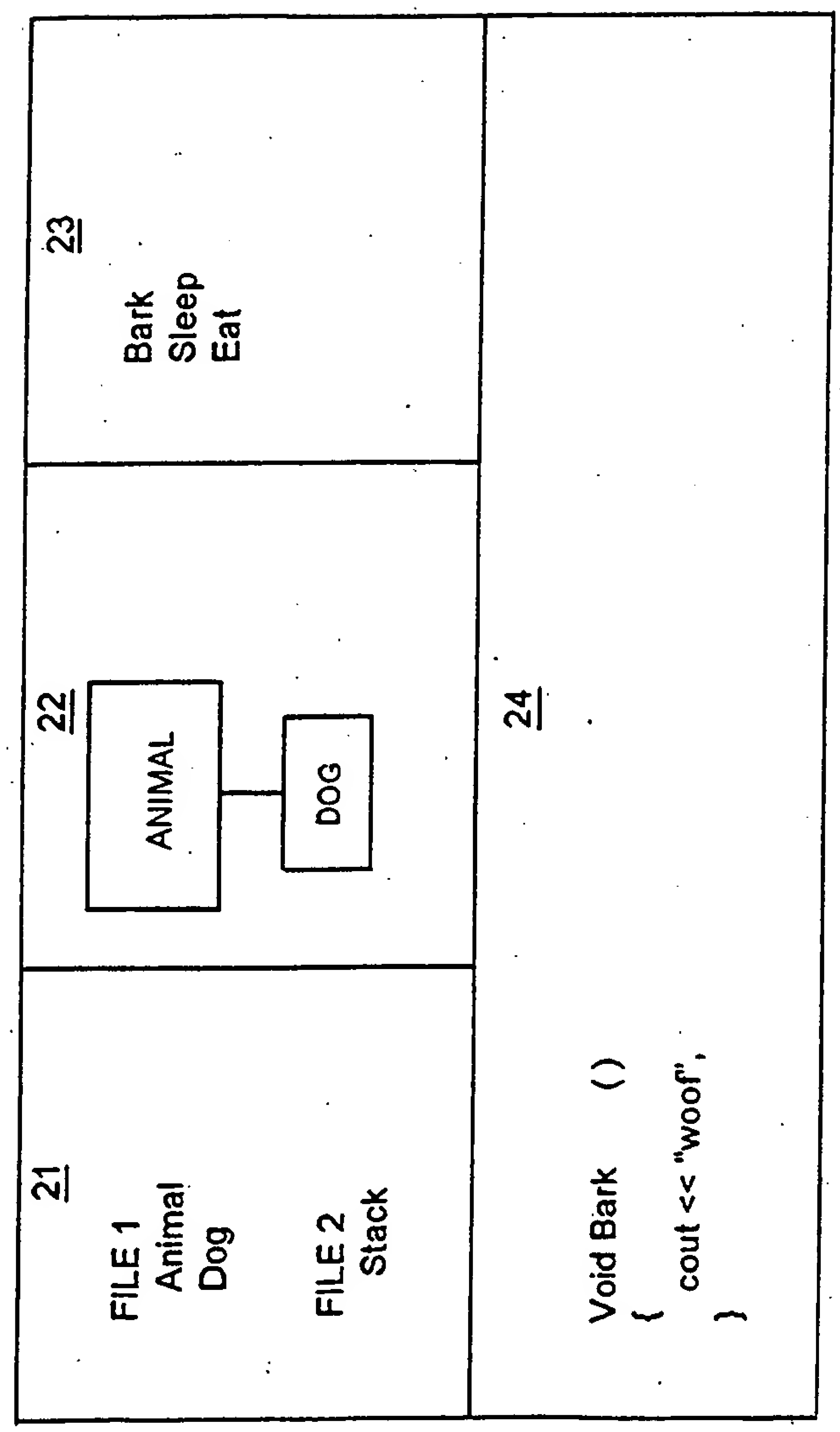
21
FILE 1
Animal
Dog
FILE 2
Stack
22
ANIMAL
DOG
23
Bark
Sleep
Eat
24
Void Bark ( )
{
cout << "woof",
}

FIG. 2

test.idl	Model Objects
#include "somobj.idl"	OBFileGroup (test.idl) OBFile (test.idl) DMFile(somobj.idl)
const short X=1;	DMDataMember(X)
enum Fruit {apple, organge};	OBEnum (Fruit) DMDataMember (apple) DMDataMember (orange)
module M1 {	OBModule(M1)
const short Y=2	DMDataMember(Y)
interface I1 {	OBInterface(I1)
typedef long x;	OBTypedef(x)
attribute string name;	DMDataMember(name)
void print(in short x); };	DMMethod(print) DMArg(x)
interface I2 {	OBinterface(I2)
struct E (char y; long x;);	OBStruct(E) DMDataMember(y) DMDataMember(x)
attribute E e1; }; };	DMDataMember (e1)

interface I1
{
}
	OBInterface(I1)

FIG. 4

2325539

REUSING CODE IN OBJECT-ORIENTED PROGRAM DEVELOPMENT

The present invention relates in general to the data processing fields and relates in particular to the field of object-oriented programming development systems.

Object-oriented development tools commonly include tools which build core portions of programming applications. Typically, a builder tool has a user interface that collects a certain amount of input from the developer for which it creates some skeleton code. This input is metadata. More generally, metadata is self-descriptive information that can describe both services and information. Typically, the generated code does not make up the entire application; thus the user/developer must extend the generated code by using a source editor. An example of such an extension is business logic that computes the salary of an employee. Having prepared the code extensions, the developer will build, that is, compile and link the application using the traditional compiler.

One problem with the traditional method is that when the metadata is changed, the user must re-generate the code. Re-generation will overwrite such code extensions, and thus the extensions need to be re-entered at each code re-generation, either by manual re-input or by cutting and pasting from another file; otherwise the editor will simply enter a null and the previously entered code extensions will be lost.

An additional problem is that traditional builders generate language-specific code. For example, one form of application wizard generates only C++ code, and thus the generated code can be used only by a C++ compiler.

Accordingly, it is an object of the present invention to provide a system that enables the persistent storage and recovery of all such edited code. It is a further object of the present invention to provide a system that enables the storage of such code extensions in a way that is independent of the language in which those code extensions are written. It is a further object of the present invention to provide a mechanism for storing edited code as metadata in support of the development of complex applications using a layered data model for the handling and storage of application development metadata. A general form of such a data model is described in Canadian Patent Application No.

Figure 5 illustrates the relationships in a preferred embodiment of the invention between an interface class and other elements of the application system.

The preferred embodiment of the invention uses the interaction between an object-oriented in-memory data model and an object-oriented persistent data model and provides application programming interfaces (APIs) to store both the interface and implementation information, including business logic, into the data model. An object builder tools helps users build business objects in an application development environment. The APIs can be invoked by user interface components as well as by components that are not part of the user interface, for example a parser or an importer. A user running the object builder tool can view and modify the contents of an interface definition language (IDL) file in the object-oriented persistent data model. From within the object builder tool the user can also edit the method implementation; the method implementation is stored in the persistent data model. If the user regenerates the source code and modifies some information in the interface definition, she can regenerate the interface and the implementation including the previously added code which has been retrieved from the object builder tools persistent data model.

Referring now to Figure 1, the overall architecture of the object-oriented persistent data model and development tools is shown according to a preferred embodiment. Element 1 includes various parsers and importers that handle previously prepared code. For example, a user can invoke the IDL parser 1 to parse an IDL file into in-memory model 4. Various builder tools 2, sometimes called SmartGuides or wizards, are forms of automatic code generating mechanisms that prompt the user or developer for input and then store the metadata in the builder tools in-memory model 4. Additional code can also be entered directly by the user, for example by the edit pane shows in Figure 2, into original files, or files prepared by parsers and importers 1 or SmartGuides 2. Code prepared by all of these methods is retained in persistent model 4. The in-memory model provides an abstraction for representing interface definition language and Java definitions, supports undo functions and queries, and isolates the parsers and importers, wizards and viewers and editors from the implementation details in object-oriented persistent data model 5. The persistent data model provides, among other things, a set of base classes for managing persistence, code generation and development builds. Preferably, each tool provides concrete

In edit pane 24, an editor window is displayed that allows the user to type in or amend the method implementation of a method that is selected in method pane 23. In the example shown here, "bark" is selected in method pane 23, and the code for the method implementation is displayed in edit pane 24. Because the method implementation is stored in the persistent data model 5 of Figure 1 according to a preferred embodiment of the invention, the action of selecting a method in method pane 23 allows the stored method implementation to be retrieved by a function of the persistent data model and displayed in edit pane 24. Whenever the source code of the application being developed is regenerated, the method implementation in edit pane 24 is saved in the persistent data model 5.

Figure 3 illustrates a group of derived classes that are utilized in implementing a preferred embodiment of the invention. All of the classes illustrated in Figure 3 are derived from the superclass DMPersistent Object 31. The class DMMethod 32 is used to model a method, and includes the attributes shown in the table below. Most of these attributes are used in ways known to the person skilled in the art, except that the attribute MethodBody is the method implementation that the user enters in an edit session shown in panel 24 in Figure 2, the MethodBody is saved as an integral part of each member of the class DMMethod and thus is recoverable at any time for regeneration of the source code. Preferably, the MethodBody is stored as a string binary large object (string blob). Saving it as a string enables the MethodBody to be recovered as a string and thus is independent of the language in which the string is expressed. The string can contain statements in any appropriate language, for example Smalltalk, C++ or Java. Therefore, no matter the environment in which the user is working, the user can save and retrieve code in the MethodBody that is appropriate to that environment.

collection of classes used in the in-memory data model, for example OBFile 42, OBModule 43, OBInterface 44, OBSimpleType 45, OBException 46, OBTypedef 48, OBEnum 50, OBStruct 52 or OBUnion 54. The class OBFileGroup 39 has a one-to-one relationship to OBFile 42 and a one-to-many relationship with each of OBModule 43, OBInterface 44, OBSimpleType 45, OBException 46, ObTypedef 48, OBEnum 50, OBStruct 52 and OBUnion 54. The relationships are ownership relationships, and when the OBFileGroup object is destructed, all of its contained elements will also be destructed.

The class DMModel 38 represents the model instance root. A user of the model attaches to a specific mode instance using methods in the class DMModel 38.

OBApplication Family 40 is a model for the objects that are saved onto a storage medium for product distribution, for example, a CD-ROM disk. These objects are then installed selectively in a target machine in a software installation step as is known in the art.

DMPart 41 is a part construct and models behaviour for other model objects by way of various relationships. The behaviour model includes, for example, inheritance, data member and method.

In a preferred embodiment of the invention, several classes derive from DMPart 41. The class OBFile 42 is used to model an IDL or Java file; each instantiation of OBFileGroup 39 contains one and only one OBFile 42. The attributes of the OBFile class are shown in the table below.

Attribute Name	Description
name	The name of the file
FileType	The type of the file. The value can be either Idl or Java.
PackageName	The name of the Package for Java
Framework	A flag to indicate whether this is a framework
OBFile::hasPublic	A flag to indicate whether this Java file contains a public interface.
OB::ForwardDecl	A blob containing the list of forward declarations for this scope.

The classes OBEnum 50, OBStruct 52 and OBUnion 54 are used to model IDL Enums Structs and Unions. The classes OBTypedef 48 OBEnum 50, OBStruct 52 and OBUnion 54 all operate through the corresponding persistent data model classes DMTypedef 47, DMEnum 49, DMStruct 51 and DMUnion 53. The classes whose names begin with "OB" in the preferred embodiment are concrete implementations of the superclasses whose names begin with "DM". The DM superclasses are not instantiated themselves.

The class DMSubpart 60 is a model type. DMDataMember 61 is a concrete implementation of DMSubPart 60. DMDataMember 61 is instantiated by the object builder tool in a preferred embodiment of the invention. DMDataMember 61 is used to model constants, attributes, struct, data members, and enums, that is enumerators. In the preferred embodiment, DMDataMember 61 includes the attributes shown in the table below.

In that table, GetImpl is a string that stores the get implementation for attributes that represent essential state of the object. SetImpl is a string that stores the set implementation for attributes that represent the essential state of the object.

Attribute Name	Description
name	The name of the corresponding item.
OB::TypeString	The type of the corresponding item. (e.g. int, or Employee).
ReadOnly	Applies only to IDL attribute. Set to 1 if it is read only. Set to 0 otherwise.
Static	A flag for Java case to indicate that static modifier
GetImpl	For attributes in IDL, and public attributes in Java, this property contains the Getter implementation.
SetImpl	For attributes in IDL, and public attributes in Java, this property contains the Setter implementation.

In the preferred embodiment of the invention described herein, each object that is created by the object builder tool in the persistent data model has a unique name. The unique name is formed by concatenating its name with its contained path. When an IDL file or a Java file is added, an OBFileGroup object representing the group, and an OBFile object representing the file are created. For example, to add "F1.idl", the system creates an OBFileGroup object of name "::F1". The "::" is

The class IXOFileGroup creates and stores the OBFileGroup object. It provides an emit method that can be called to generate a code for the interface and implementation that are associated with a specific file group.

The class IXOFile creates and stores the OBFile object that represents an IDL or Java file. It also creates an DMFile object for each include or import statement. It provides methods to add and to query module, interface, constant, typedef, struct, enum, union, and exception that are scoped at the specific file level.

The class IXOModule creates and it stores the OBModule object that represents an IDL module. It provides methods to add and query module, interface, constant, typedef, struct, enum, union, and exception that are scoped at the specific module level. The class IXOInterface creates and stores the OBInterface object that represents an IDL interface or a Java interface. This class provides methods to add, query and find attribute, method, constant, typedef, struct, enum, union, and exception that are scoped at the specific interface level. The class IXOInterface also provides methods to add and to query the parent interfaces of the specific interface. When a parent interface is added, any method that needs to overwritten or implemented from the parent class will result in a DMMethod object being created and added to this class.

The class IXOType encapsulates a type definition and an IXOType object is created for each of the following cases or events: the type of an attribute, the return type of a method, the type of a parameter, the type of a struct data member, the type of an exception data member, the type of a union data member and the type of a TypeDef. In order to add any of these objects to the persistent data model, the DMPart object representing its type must already have been added to the persistent model. The IXOType class provides operations to locate the DMPart that represents the type, or to add an empty OBInterface object that represents the type.

The class IXOMember creates and stores the DMDataMember and associates it with a DMPart.

The class IXOConstant is derived from IXOMember. It is used to represent an IDL constant. Each IDL constant has a name, a type, and a value. This class called a method IXOMember::to cdm to create the

```
}
```

When the user decides to make the change permanent, for example by clicking on the OK button in a user dialogue box, the DMMethod including the MethodBody is saved in the persistent data model.

When the user makes the change to add a parameter at a later date, for example, to change the print statement from having no parameters to having the parameter in short p1, the MethodBody is not changed by the system.

The generated a.IDL file would include the code below:

```
interface A
    {
        void print (in short p1);
    }
```

Thus the text that the user previously entered is recovered in the form below for viewing and for further editing at the option of the user.

```
void A::print(in short p1)
    {
        cout << "hello";
    }
```

As described above, the MethodBody is saved as a string binary large object, and therefore is not changed by the system except upon being edited by the user. Because the MethodBody is a string, it can contain material that can be used in any platform upon which the user is working. For example, it can contain Java code or C++ code or a combination of text and code an individual language.

An advantage of the invention is that the user does not have to re-enter large amounts of text and code from IDL files that previously would overwrite text that was entered by way of an editor. A further advantage is that the invention is inherently independent of the language used because the code and text entered by means of an editor is stored as string data in the MethodBody attribute of the instances of a DMMethod class, and is retrieved in the same way.

5. A system as claimed in Claim 2, 3 or 4 wherein said implementation information is stored in the form of a binary large object block.

6. A system as claimed in Claim 4 wherein said metadata is stored as method implementation information.

7. A system as claimed in any of Claims 4 to 6 wherein said implementation information comprises business logic.

8. A system as claimed in any of Claims 4 to 7 wherein said implementation information comprises a text string.

9. A system as claimed in any of Claims 4 to 8 wherein said implementation information comprises a block of code in a high level computer language.